Engenharia de Software: do Requisito ao Projeto - Princípios SOLID

Exercícios

1) Considere o código do quadro abaixo.

```
public class GerenciadorArquivos {
    private String nomeArquivo;

    public GerenciadorArquivos(String nomeArquivo) {
        this.nomeArquivo = nomeArquivo;
    }

    public void lerArquivo() {
        try (BufferedReader br = new BufferedReader(new FileReader(nomeArquivo))) {
            String linha;
            int totalLinhas = 0;
            int totalPalavras = 0;
            int totalCaracteres = 0;

            while ((linha = br.readLine()) != null) {
                totalLinhas++;
                totalPalavras += linha.split(" ").length;
                totalCaracteres += linha.length();
            }

            System.out.println("Total de linhas: " + totalLinhas);
            System.out.println("Total de palavras: " + totalPalavras);
            System.out.println("Total de caracteres: " + totalCaracteres);
        } catch (IOException e) {
            System.err.println("Erro ao ler o arquivo: " + e.getMessage());
        }
    }

    public void processarDados() {
        // Lógica para processar os dados do arquivo
    }
}
```

a. Descreva qual princípio SOLID o código viola e justifique a sua resposta.

Princípio violado: Princípio da Responsabilidade Única (Single Responsibility Principle - SRP).

Justificativa: A classe GerenciadorArquivos acumula múltiplas responsabilidades que deveriam ser separadas. Uma classe deve ter apenas um motivo para mudar. No entanto, esta classe mudaria por diversos motivos:

1. Leitura do arquivo: Se a forma de ler o arquivo mudar (ex: ler de um recurso de rede em vez de um arquivo local).
2. Análise dos dados: Se a lógica para contar palavras ou caracteres for alterada (ex: considerar pontuação de forma diferente).
3. Apresentação dos dados: Se o formato de exibição dos resultados mudar (ex: salvar em um log, exibir em uma interface gráfica em vez do console).
4. Processamento dos dados: A existência do método processarDados() indica uma quarta responsabilidade distinta.

Por acumular as responsabilidades de leitura, análise, apresentação e processamento, a classe viola o SRP, tornando-se menos coesa, mais difícil de manter e de testar.

b. Refatore o código para que se adeque ao princípio violado.

Para adequar o código ao SRP, vamos separar as responsabilidades em classes distintas.

1. LeitorDeArquivo: Responsável apenas por ler o conteúdo do arquivo.
2. AnalisadorDeConteudo: Responsável por analisar o conteúdo lido e extrair as estatísticas.
3. ExibidorDeResultados: Responsável por formatar e exibir os resultados da análise.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

// Classe para conter os resultados da análise
class ResultadoAnalise {
    private final int totalLinhas;
    private final int totalPalavras;
    private final int totalCaracteres;

    public ResultadoAnalise(int totalLinhas, int totalPalavras, int totalCaracteres) {
        this.totalLinhas = totalLinhas;
        this.totalPalavras = totalPalavras;
        this.totalCaracteres = totalCaracteres;
    }

    // Getters para os resultados
    public int getTotalLinhas() { return totalLinhas; }
    public int getTotalPalavras() { return totalPalavras; }
    public int getTotalCaracteres() { return totalCaracteres; }
}

// Responsabilidade 1: Ler o arquivo
class LeitorDeArquivo {
    private final String nomeArquivo;

    public LeitorDeArquivo(String nomeArquivo) {
        this.nomeArquivo = nomeArquivo;
    }
```

```
    public List<String> lerLinhas() throws IOException {
        List<String> linhas = new ArrayList<>();
        try (BufferedReader br = new BufferedReader(new FileReader(nomeArquivo))) {
            String linha;
            while ((linha = br.readLine()) != null) {
                linhas.add(linha);
            }
        }
        return linhas;
    }
}

// Responsabilidade 2: Analisar o conteúdo
class AnalisadorDeConteudo {
    public ResultadoAnalise analisar(List<String> linhas) {
        int totalLinhas = linhas.size();
        int totalPalavras = 0;
        int totalCaracteres = 0;

        for (String linha : linhas) {
            totalPalavras += linha.split("\\s+").length; // Usa regex para contar palavras
            totalCaracteres += linha.length();
        }
        return new ResultadoAnalise(totalLinhas, totalPalavras, totalCaracteres);
    }
}

// Responsabilidade 3: Exibir os resultados
class ExibidorDeResultados {
    public void exibir(ResultadoAnalise resultado) {
        System.out.println("Total de linhas: " + resultado.getTotalLinhas());
        System.out.println("Total de palavras: " + resultado.getTotalPalavras());
        System.out.println("Total de caracteres: " + resultado.getTotalCaracteres());
    }
}

// Classe principal para orquestrar as operações
public class Main {
    public static void main(String[] args) {
        String nomeArquivo = "meu_arquivo.txt"; // Exemplo de nome de arquivo
        try {
            LeitorDeArquivo leitor = new LeitorDeArquivo(nomeArquivo);
            List<String> conteudo = leitor.lerLinhas();

            AnalisadorDeConteudo analisador = new AnalisadorDeConteudo();
            ResultadoAnalise resultado = analisador.analisar(conteudo);

            ExibidorDeResultados exibidor = new ExibidorDeResultados();
            exibidor.exibir(resultado);

        } catch (IOException e) {
            System.err.println("Erro ao processar o arquivo: " + e.getMessage());
        }
    }
```

```
}
```

2) Considere o código do quadro abaixo.

```
public class CalculadoraImpostos {
    public double calcularImposto(double valor, String tipoImposto) {
        double imposto = 0.0;

        if (tipoImposto.equals("ICMS")) {
            imposto = valor * 0.18; // Taxa de ICMS
        } else if (tipoImposto.equals("ISS")) {
            imposto = valor * 0.05; // Taxa de ISS
        } else if (tipoImposto.equals("IPI")) {
            imposto = valor * 0.10; // Taxa de IPI
        } else if (tipoImposto.equals("IOF")) {
            imposto = valor * 0.03; // Taxa de IOF
        }

        return imposto;
    }
}
```

a. Descreva qual princípio SOLID o código viola e justifique a sua resposta.

Princípio violado: Princípio Aberto/Fechado (Open/Closed Principle - OCP).

Justificativa: O princípio OCP afirma que uma entidade de software deve ser aberta para extensão, mas fechada para modificação. A classe CalculadoraImpostos viola essa regra porque, toda vez que um novo tipo de imposto precisa ser adicionado (ex: PIS, COFINS), é necessário modificar o código existente, adicionando um novo bloco else if ao método calcularImposto. O ideal seria poder adicionar novos impostos (estender o comportamento) sem alterar o código já existente e testado.

b. Refatore o código para que se adeque ao princípio violado.

Para adequar o código ao OCP, podemos usar o Padrão de Projeto Strategy. Criaremos uma interface para a estratégia de cálculo de imposto e uma classe concreta para cada tipo de imposto.

```
// Interface Strategy
public interface CalculoImposto {
    double calcular(double valor);
}

// Estratégias Concretas para cada imposto
class CalculoICMS implements CalculoImposto {
    @Override
    public double calcular(double valor) {
        return valor * 0.18; // Taxa de ICMS
    }
```

```
}

class CalculoISS implements CalculoImposto {
    @Override
    public double calcular(double valor) {
        return valor * 0.05; // Taxa de ISS
    }
}

class CalculoIPI implements CalculoImposto {
    @Override
    public double calcular(double valor) {
        return valor * 0.10; // Taxa de IPI
    }
}

class CalculoIOF implements CalculoImposto {
    @Override
    public double calcular(double valor) {
        return valor * 0.03; // Taxa de IOF
    }
}

// A calculadora agora usa uma estratégia para fazer o cálculo
public class CalculadoraImpostos {
    public double calcularImposto(double valor, CalculoImposto estrategiaDeCalculo) {
        return estrategiaDeCalculo.calcular(valor);
    }
}

// Exemplo de uso
public class Main {
    public static void main(String[] args) {
        CalculadoraImpostos calculadora = new CalculadoraImpostos();
        double valorBase = 1000.0;

        // Calcula o ICMS
        double icms = calculadora.calcularImposto(valorBase, new CalculoICMS());
        System.out.println("ICMS a ser pago: " + icms); // 180.0

        // Calcula o ISS
        double iss = calculadora.calcularImposto(valorBase, new CalculoISS());
        System.out.println("ISS a ser pago: " + iss); // 50.0

        // Para adicionar um novo imposto, basta criar uma nova classe que implementa
CalculoImposto.
        // Nenhuma modificação na classe CalculadoraImpostos é necessária.
    }
}
```

3) Considere o código do quadro abaixo.

```
class Retangulo {
    protected int largura;
    protected int altura;

    public void setLargura(int largura) {
        this.largura = largura;
    }

    public void setAltura(int altura) {
        this.altura = altura;
    }

    public int calcularArea() {
        return largura * altura;
    }
}

class Quadrado extends Retangulo {
    @Override
    public void setLargura(int lado) {
        super.setLargura(lado);
        super.setAltura(lado);
    }

    @Override
    public void setAltura(int lado) {
        super.setLargura(lado);
        super.setAltura(lado);
    }
}

public class Main {
    public static void main(String[] args) {
        Retangulo retangulo = new Quadrado();
        retangulo.setLargura(5);
        retangulo.setAltura(10);

        System.out.println("Área do retângulo: " + retangulo.calcularArea());
    }
}
```

a. Descreva qual princípio SOLID o código viola e justifique a sua resposta.

Princípio violado: Princípio da Substituição de Liskov (Liskov Substitution Principle - LSP).

Justificativa: O LSP estabelece que objetos de uma classe derivada devem poder substituir objetos de sua classe base sem quebrar o funcionamento do programa. Neste caso, um Quadrado não é um substituto adequado para um Retangulo.

A classe base Retangulo tem um contrato implícito: alterar sua largura não deve afetar sua altura, e vice-versa. A classe derivada Quadrado quebra esse contrato, pois ao chamar setLargura() ou setAltura(), ambos os lados são modificados.

O código main demonstra a quebra de expectativa:

1. retangulo.setLargura(5); // Espera-se um retângulo com largura 5.
2. retangulo.setAltura(10); // Espera-se um retângulo com largura 5 e altura 10.
3. retangulo.calcularArea(); // O resultado esperado seria 5 * 10 = 50.

No entanto, como retangulo é na verdade um Quadrado, a segunda chamada (setAltura(10)) também altera a largura para 10. O resultado do cálculo será 10 * 10 = 100, quebrando a lógica esperada pelo cliente que manipula um Retangulo.

b. Refatore o código para que se adeque ao princípio violado.

A relação de herança "Quadrado é um Retângulo" é matematicamente correta, mas problemática em um modelo de objetos com estado mutável. A solução é quebrar essa herança e, em vez disso, usar uma abstração comum, como uma interface FormaGeometrica.

```
// Abstração comum
public interface FormaGeometrica {
    int calcularArea();
}

// Classe Retangulo, sem relação de herança com Quadrado
class Retangulo implements FormaGeometrica {
    protected int largura;
    protected int altura;

    public void setLargura(int largura) {
        this.largura = largura;
    }

    public void setAltura(int altura) {
        this.altura = altura;
    }

    @Override
    public int calcularArea() {
        return largura * altura;
    }
}

// Classe Quadrado, implementa a interface mas não herda de Retangulo
class Quadrado implements FormaGeometrica {
```

```
    private int lado;

    public void setLado(int lado) {
        this.lado = lado;
    }

    @Override
    public int calcularArea() {
        return lado * lado;
    }
}

// Exemplo de uso
public class Main {
    public static void main(String[] args) {
        Retangulo retangulo = new Retangulo();
        retangulo.setLargura(5);
        retangulo.setAltura(10);
        System.out.println("Área do retângulo: " + retangulo.calcularArea()); // 50

        Quadrado quadrado = new Quadrado();
        quadrado.setLado(5);
        System.out.println("Área do quadrado: " + quadrado.calcularArea()); // 25
    }
}
```

4) Considere o código do quadro abaixo.

```
interface Documento {
    void criar();
    void visualizar();
    void editar();
    void imprimir();
}

class DocumentoTexto implements Documento {
    @Override
    public void criar() {
        System.out.println("Documento de texto criado.");
    }

    @Override
    public void visualizar() {
        System.out.println("Documento de texto visualizado.");
    }

    @Override
    public void editar() {
        System.out.println("Documento de texto editado.");
    }

    @Override
    public void imprimir() {
        System.out.println("Documento de texto impresso.");
    }
}

class DocumentoPDF implements Documento {
    @Override
    public void criar() {
        System.out.println("PDF criado.");
    }

    @Override
    public void visualizar() {
        System.out.println("PDF visualizado.");
    }

    @Override
    public void editar() {
        // Não aplicável para documentos PDF
    }

    @Override
```

```
    public void imprimir() {
        System.out.println("PDF impresso.");
    }
}

public class Main {
    public static void main(String[] args) {
        Documento documentoPDF = new DocumentoPDF();
        documentoPDF.criar();
        documentoPDF.visualizar();
        documentoPDF.imprimir();
    }
}
```

a. Descreva qual princípio SOLID o código viola e justifique a sua resposta.

Princípio violado: Princípio da Segregação de Interface (Interface Segregation Principle - ISP).

Justificativa: O ISP afirma que um cliente (neste caso, uma classe que implementa uma interface) não deve ser forçado a depender de métodos que não utiliza. A interface Documento é uma "interface gorda" (fat interface), pois agrupa múltiplos comportamentos (criar, visualizar, editar, imprimir).

A classe DocumentoPDF é forçada a implementar o método editar(), mesmo que essa operação não seja aplicável para ela ("Não aplicável para documentos PDF"). Isso resulta em uma implementação vazia ou que lança uma exceção, o que é um claro sinal de violação do ISP. As interfaces devem ser coesas e específicas para as necessidades de seus clientes.

b. Refatore o código para que se adeque ao princípio violado.

Para adequar o código ao ISP, devemos "segregar" ou dividir a interface Documento em interfaces menores e mais específicas, baseadas em suas capacidades.

```
// Interfaces segregadas baseadas em capacidades

// Capacidade de ser criado e visualizado
interface Documento {
    void criar();
    void visualizar();
}

// Capacidade de ser editado
interface Editavel {
    void editar();
}
```

```
// Capacidade de ser impresso
interface Imprimivel {
    void imprimir();
}

// DocumentoTexto implementa todas as capacidades
class DocumentoTexto implements Documento, Editavel, Imprimivel {
    @Override
    public void criar() {
        System.out.println("Documento de texto criado.");
    }
    @Override
    public void visualizar() {
        System.out.println("Documento de texto visualizado.");
    }
    @Override
    public void editar() {
        System.out.println("Documento de texto editado.");
    }
    @Override
    public void imprimir() {
        System.out.println("Documento de texto impresso.");
    }
}

// DocumentoPDF não implementa a capacidade de edição
class DocumentoPDF implements Documento, Imprimivel {
    @Override
    public void criar() {
        System.out.println("PDF criado.");
    }
    @Override
    public void visualizar() {
        System.out.println("PDF visualizado.");
    }
    @Override
    public void imprimir() {
        System.out.println("PDF impresso.");
    }
}

// Exemplo de uso
public class Main {
    public static void main(String[] args) {
        DocumentoTexto docTexto = new DocumentoTexto();
        docTexto.criar();
        docTexto.editar();
        docTexto.imprimir();

        System.out.println("---");

        DocumentoPDF docPDF = new DocumentoPDF();
        docPDF.criar();
        docPDF.imprimir();
```

```
        // A chamada docPDF.editar() agora causa um erro de compilação,
        // o que é correto, pois a classe não expõe esse método.
    }
}
```

5) Considerando os exercícios 1 a 6 do capítulo 9 faça:

a) Identifique possíveis violações dos Princípios SOLID e justifique porque os trechos de códigos ou classes violam o(s) princípio(s) identificados.

b) Refatore os trechos de códigos ou classes de forma que estes não violem os princípios identificados.

Exercício 1 e 2: Casos de Uso "Calcula Data de Devolução" e "Emprestar Livro"

Estes exercícios pedem para aplicar os padrões GRASP Expert e Creator para implementar as funcionalidades de um empréstimo. A violação SOLID ocorreria em uma implementação que não seguisse esses padrões.

Princípio Violado: Princípio da Responsabilidade Única (SRP).

Justificativa: Uma abordagem ingênua seria criar uma única classe massiva, como SistemaBiblioteca ou ControladorGeral, que implementaria todo o fluxo do caso de uso "Emprestar Livro":

1. Verificar se o Aluno está cadastrado.
2. Verificar se o Aluno possui débitos.
3. Verificar a disponibilidade de cada Livro.
4. Criar o objeto Emprestimo.
5. Criar os objetos ItemEmprestimo.
6. Calcular a data de devolução.
7. Gravar tudo no banco de dados.

Essa classe teria múltiplas e distintas razões para mudar, violando grosseiramente o SRP. Qualquer alteração na regra de cálculo de data, na verificação de débitos ou no cadastro de alunos exigiria a modificação dessa mesma classe, tornando-a frágil e complexa.

A refatoração consiste em distribuir as responsabilidades para as classes especialistas na informação (GRASP Expert), o que naturalmente nos leva a um design que respeita o SRP.

1. Cálculo da Data de Devolução:
    - A classe Emprestimo deve ser responsável por orquestrar o cálculo, mas a informação do prazo está na classe Titulo.

    - Solução: A classe Emprestimo pode ter um método calcularDataDevolucao(). Este método percorreria seus ItemEmprestimos, pediria o Titulo de cada um e, a partir do prazo do Titulo, calcularia a data final. Isso faz com que cada classe seja responsável por aquilo que conhece: Titulo conhece seu prazo, Emprestimo conhece seus itens e a data de início.

2. Criação de Empréstimo:
    - Seguindo o GRASP Creator, a classe Aluno é uma forte candidata a criar um Emprestimo, pois ela "agrega" ou está intimamente associada aos empréstimos.

- Solução: Aluno pode ter um método realizarEmprestimo(List<Livro> livros). Dentro deste método, ele instancia um Emprestimo, associa a si mesmo e delega ao Emprestimo a criação dos ItemEmprestimo. Isso aumenta a coesão da classe Aluno (ela gerencia suas próprias operações) e evita sobrecarregar um controlador genérico.

Exercício 3 e 4: Coesão e Acoplamento na Devolução de Livro

Estes exercícios focam em coesão e acoplamento, analisando as figuras 9.18-9.20 para a devolução de um livro (que poderia gerar um débito/multa).

Princípios Violados: Princípio da Inversão de Dependência (DIP) e Princípio da Responsabilidade Única (SRP).

Justificativa: A solução problemática (semelhante à Figura 9.18) seria ter uma classe de alto nível (como ControladorDevolucao) diretamente responsável por verificar um empréstimo, calcular a multa e criar um objeto Debito.

- Violação de SRP: O controlador estaria fazendo mais do que apenas coordenar; estaria executando a lógica de negócio de cálculo de multa, que não lhe pertence.

- Violação de DIP: A classe de alto nível ControladorDevolucao estaria dependendo diretamente de uma classe de baixo nível Debito (especificamente, da sua criação new Debito()). O correto seria que ambos dependessem de abstrações.

A solução correta, alinhada com as figuras 9.19 e 9.22, é delegar a responsabilidade.

1. A classe Aluno recebe a solicitação de devolver(codLivro).

2. Aluno não sabe os detalhes. Ele encontra o Emprestimo correspondente e delega: emprestimo.realizarDevolucao().

3. A classe Emprestimo é a Expert para saber se está atrasado. Se estiver, ela mesma (GRASP Creator) cria a instância de Debito, pois possui todas as informações necessárias para isso (data de devolução vs. data prevista).

4. O Controlador apenas orquestra a chamada inicial para aluno.devolver(), sem conhecer Debito ou as regras de cálculo. Isso resulta em um design com baixo acoplamento, alta coesão e que respeita SRP e DIP.

Exercício 5 e 6: Padrão Controlador e a Classe CBiblioteca

Estes exercícios abordam a aplicação do padrão Controlador, incluindo a análise da Figura 9.26, que mostra uma classe CBiblioteca como um controlador central (Facade).

Princípios Violados: Princípio da Segregação de Interface (ISP) e Princípio da Responsabilidade Única (SRP).

Justificativa: Um controlador centralizado como CBiblioteca (padrão Facade) para um sistema com múltiplos casos de uso complexos pode se tornar um "God Object" (Objeto Divino).

- Violação de SRP: Esta classe seria responsável por coordenar empréstimos, devoluções, cadastros de usuários, pesquisas, etc. Ela teria muitas razões para mudar.
- Violação de ISP: A interface exposta por CBiblioteca seria imensa. Um cliente (como uma tela de cadastro de livros) que precisa apenas do método cadastrarLivro() seria forçado a depender de uma interface que também contém realizarEmprestimo(), pagarDebito(), etc., métodos que ele nunca usará.

A refatoração é abandonar o modelo de um único controlador de fachada e adotar o padrão Controlador por Caso de Uso.

1. Dividir o Controlador: Em vez de uma CBiblioteca, crie classes menores e mais focadas:
    - EmprestarLivroController
    - DevolverLivroController
    - CadastrarAlunoController

2. Interfaces Específicas: Cada controlador implementaria uma interface específica para sua função. A interface do usuário (GUI) para realizar um empréstimo dependeria apenas de EmprestarLivroController, recebendo uma interface limpa e coesa que atende exatamente às suas necessidades.

3. Benefícios: Esta abordagem respeita o SRP, pois cada controlador tem uma única responsabilidade (coordenar um caso de uso). Respeita o ISP, pois os clientes dependem de interfaces pequenas e específicas. E também respeita o Princípio Aberto/Fechado (OCP), pois para adicionar um novo caso de uso, basta criar um novo controlador sem modificar os existentes.